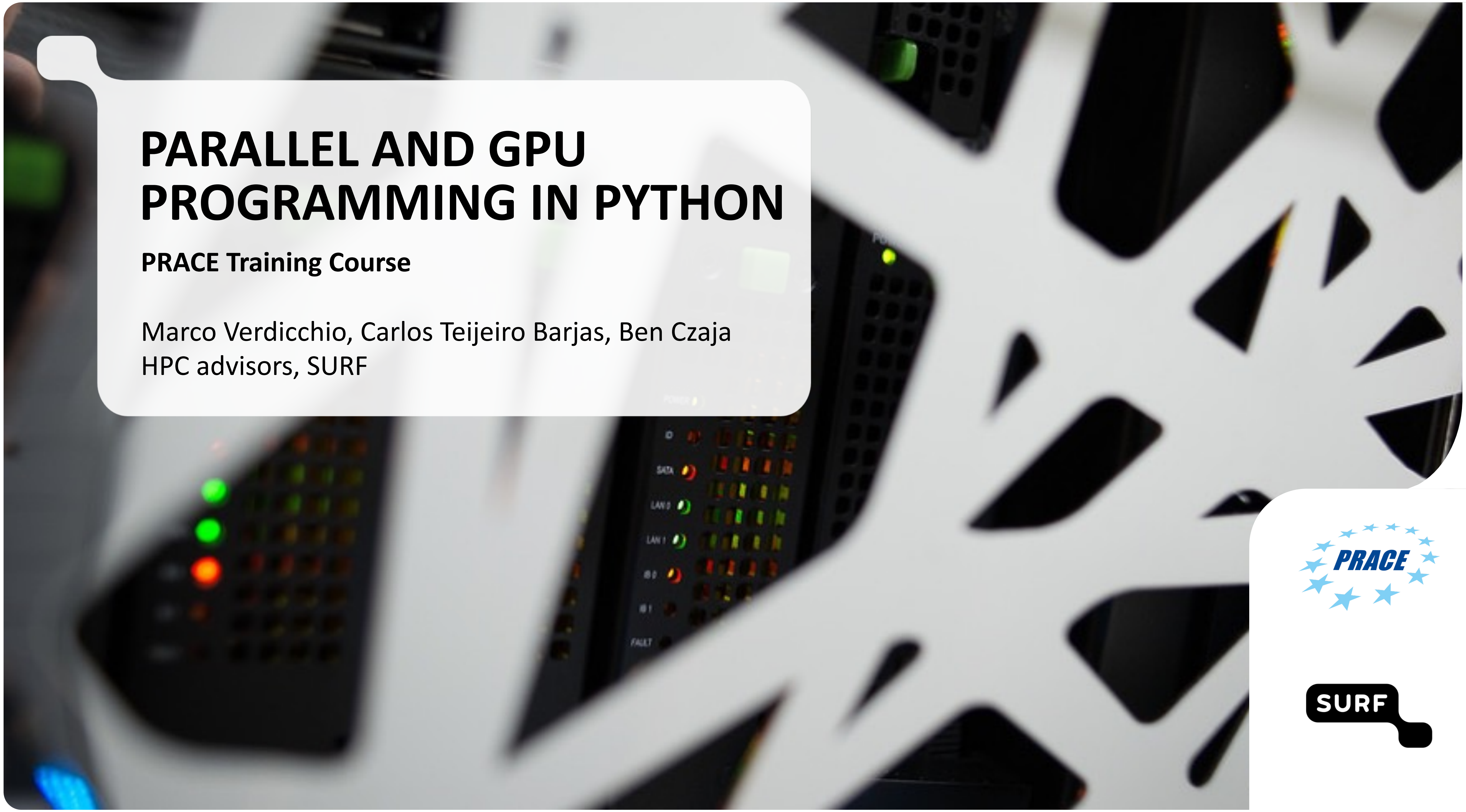


PARALLEL AND GPU PROGRAMMING IN PYTHON

PRACE Training Course

Marco Verdicchio, Carlos Teijeiro Barjas, Ben Czaja
HPC advisors, SURF

Recap Day 1

Parallel computing models

- Task parallel
  - many independent runs
  - needs orchestration
  - for monte-carlo, parameter sweeps
- Shared memory
  - always within one batch node
  - uses threads
  - often implicit
- Distributed memory
  - can use one or more batch nodes
  - uses separate processes
  - almost always using MPI
  - for PDE problems, time stepping

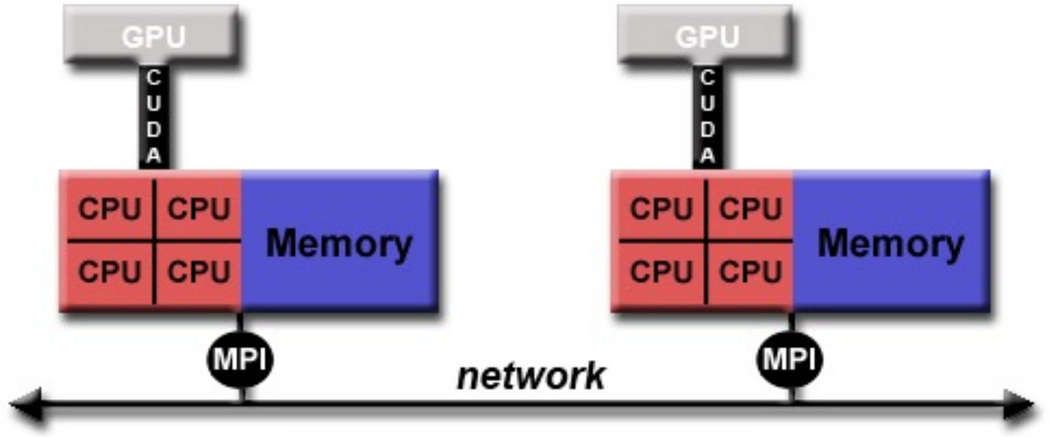


Image source: computing.llnl.gov

PTC: Parallel and GPU programming in Python

DAY 2

Start	End	Duration	Title
09:00	10:00	01:00	Introduction to GPU computing
10:00	10:15	00:15	Coffee break
10:15	11:15	01:00	GPU programming with Python
11:15	12:15	01:00	Hands-on: GPU programming with Python
12:15	13:15	01:00	Lunch
13:15	14:45	01:30	Advanced GPU computing with Python
14:45	15:00	00:15	Coffee break
15:00	17:00	02:00	Hands-on: Advanced GPU computing with Python

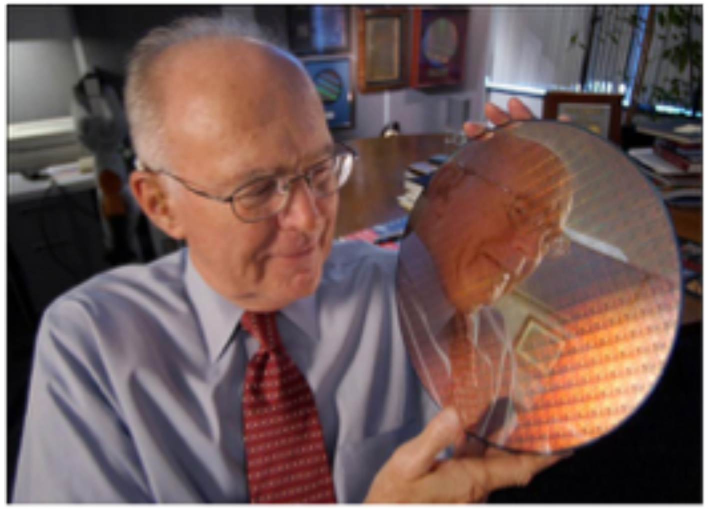
Introduction to GPU computing

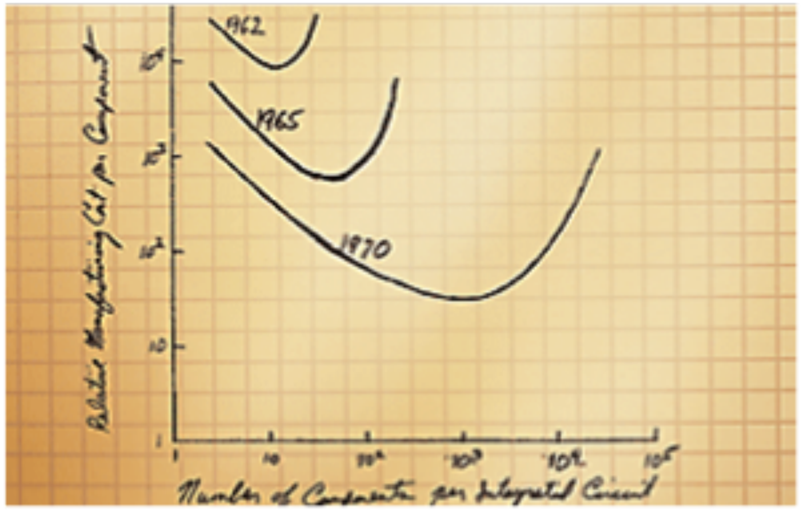
Moore’s law

- Gordon Moore (co-founder of Intel) predicted in 1965 that the transistor density of semiconductor chips would double roughly every 18 months.

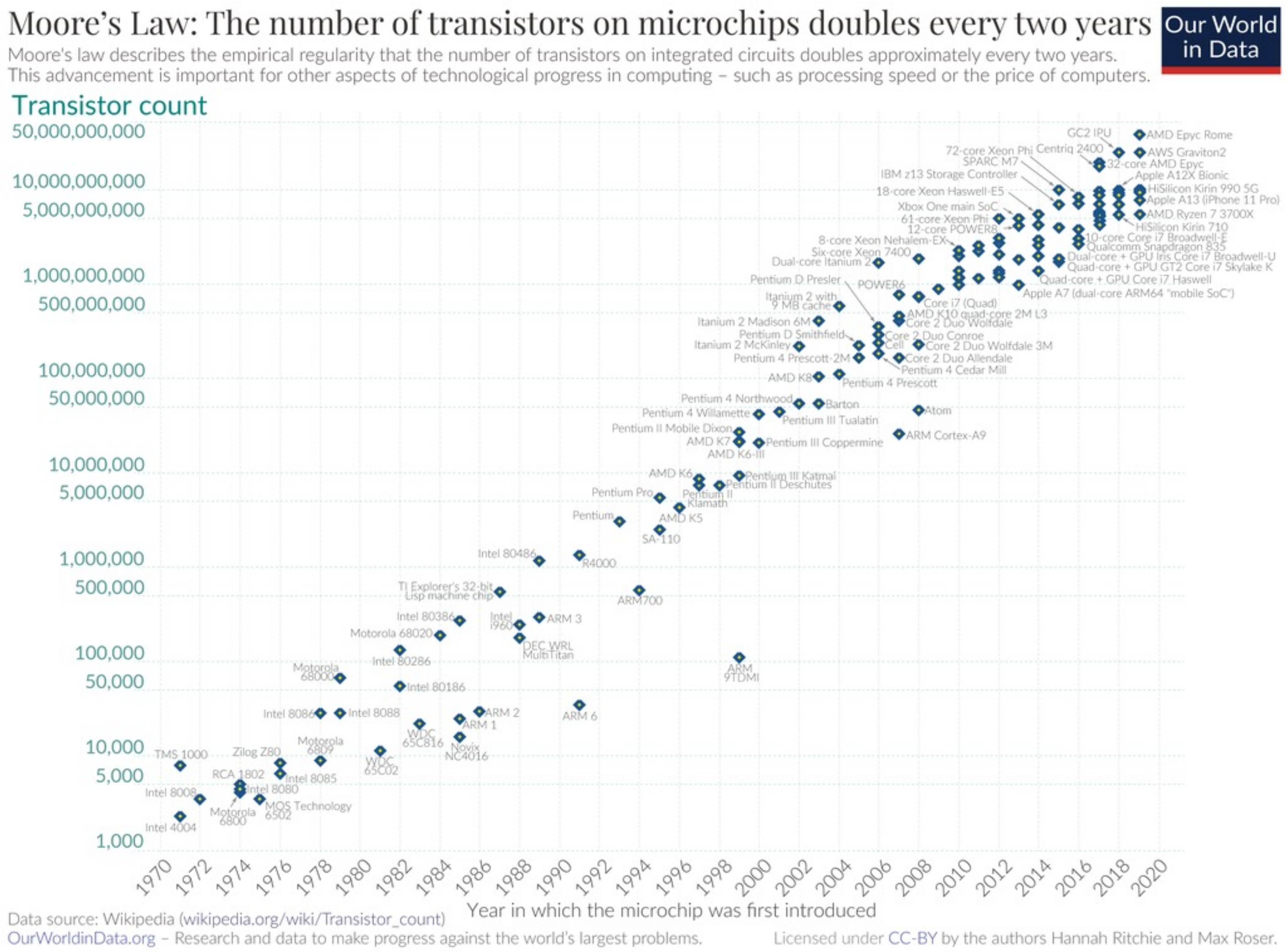
- “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year ... Certainly over the short term this rate can be expected to continue, if not to increase....”

  Electronics Magazine 1965




1962
1965
1970
Relative Manufacturing Cost per Component
10^4
10^3
10^2
10
1
1
10
10^2
10^3
10^4
10^5
Number of Components per Integrated Circuit

Introduction to GPU computing



Moore’s Law: The number of transistors on microchips doubles every two years

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

OurWorldinData.org – Research and data to make progress against the world’s largest problems.

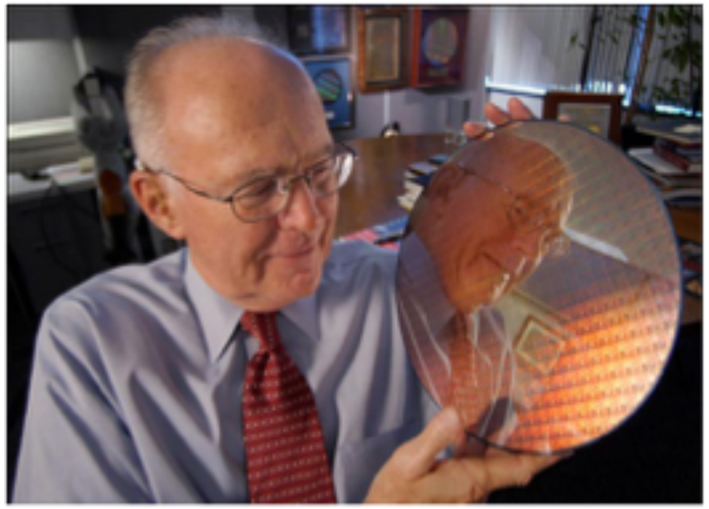
Licensed under CC-BY by the authors Hannah Ritchie and Max Roser.

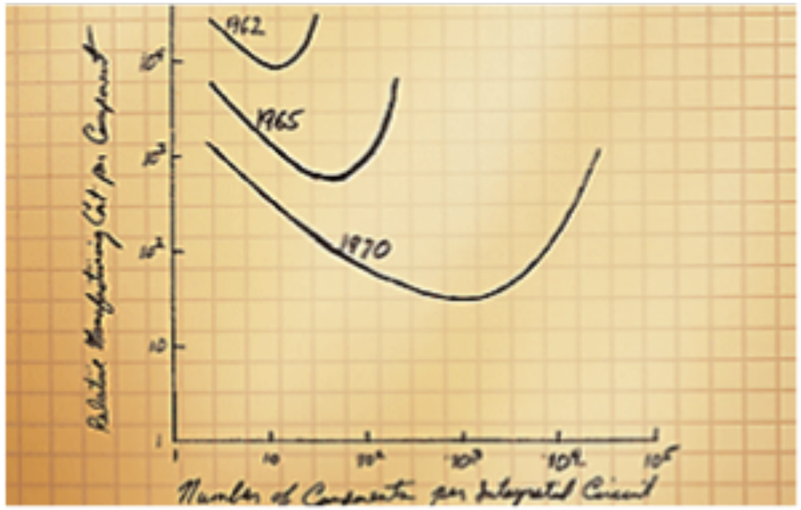
Introduction to GPU computing

Moore's law

- More transistors = more functionality
- Improved technology = faster clocks = more speed
- Thus, every 18 months, we obtained better and faster processors.
- They were all sequential: they execute one operation per clock cycle.

- We no longer gain performance by "growing" sequential processors




1962
1965
1970
Relative Manufacturing Cost per Component
10^4
10^3
10^2
10
1
1
10
10^2
10^3
10^4
10^5
Number of Components per Integrated Circuit

Introduction to GPU computing

New ways of using transistors

Improve PERFORMANCE by using parallelism on-chip:
multi-core (CPUs) and many-core processors (GPUs).

Introduction to GPU computing

Graphic Processing Units (GPUs)

GPU is one of the main hardware components of current computer architecture (e.g: phones, game consoles, HPC!)

Originally designed to render images to display, modern graphic processing units are used for some of the most complex calculations, such as big data, ML/DL and AI.

Within the years GPUs evolved from single core, fixed functional hardware, into a set of programmable parallel cores.

This started the era of General Purpose GPU (GPGPU).

Introduction to GPU computing

Graphic Processing Units (GPUs)

History

- 1990's real time 3D rendering (Video games, Movies, etc.)
  - Super VGA (SVGA) or even Ultra VGA (UVGA)
- Computationally very expensive!!!
- Before 1990's graphics were done on CPU's as well as framerate:
  - reduced quality images
  - No 3D rendering

Introduction to GPU computing

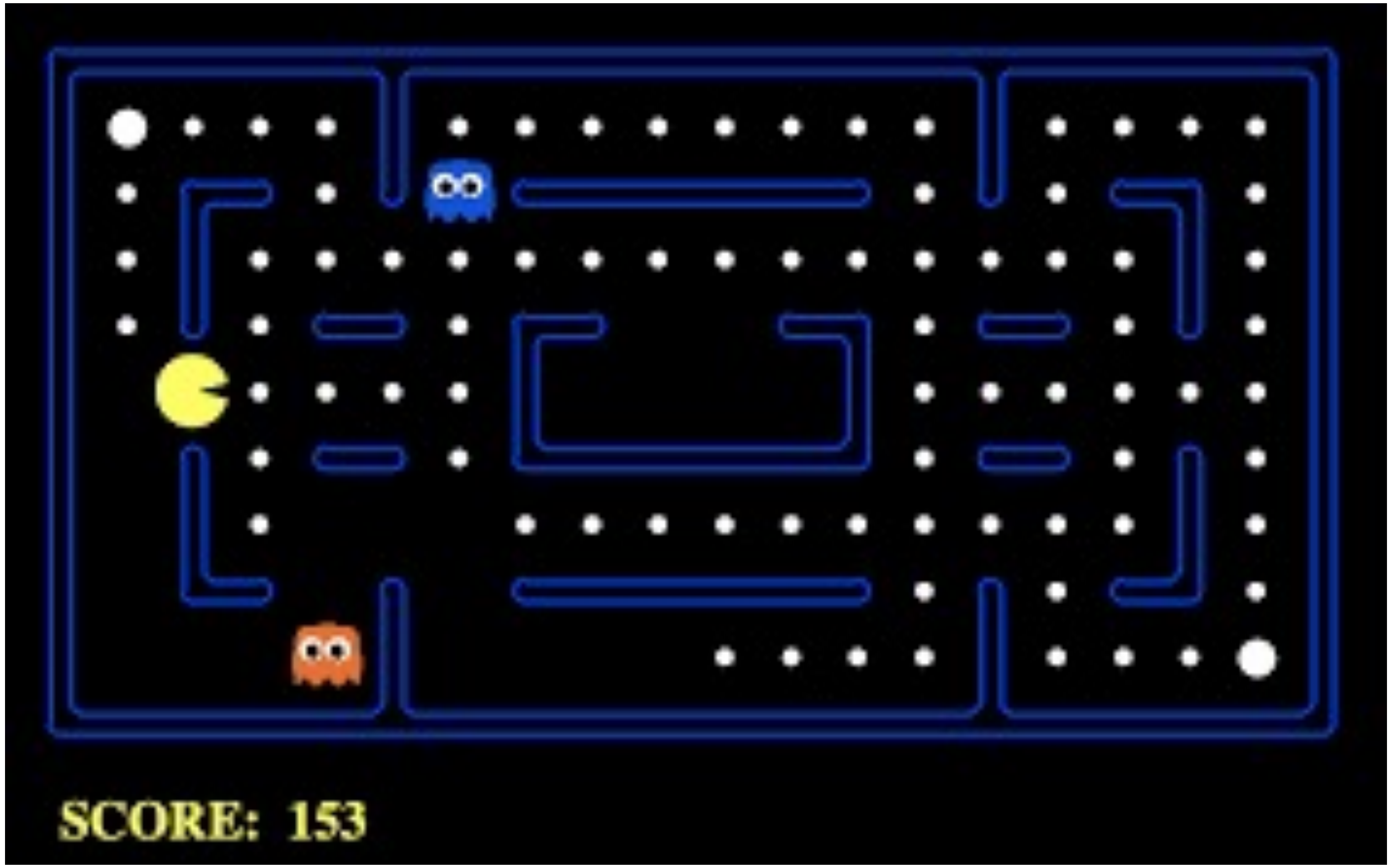
Graphic Processing Units (GPUs)

History

- 1996 introduction Voodoo graphic chip
  - one of the first video card for parallel work
- NVIDIA 1997 released the first chip to combine 3D acceleration with traditional 2D and video acceleration.
- 1999 first “GPU” with the GeForce 256

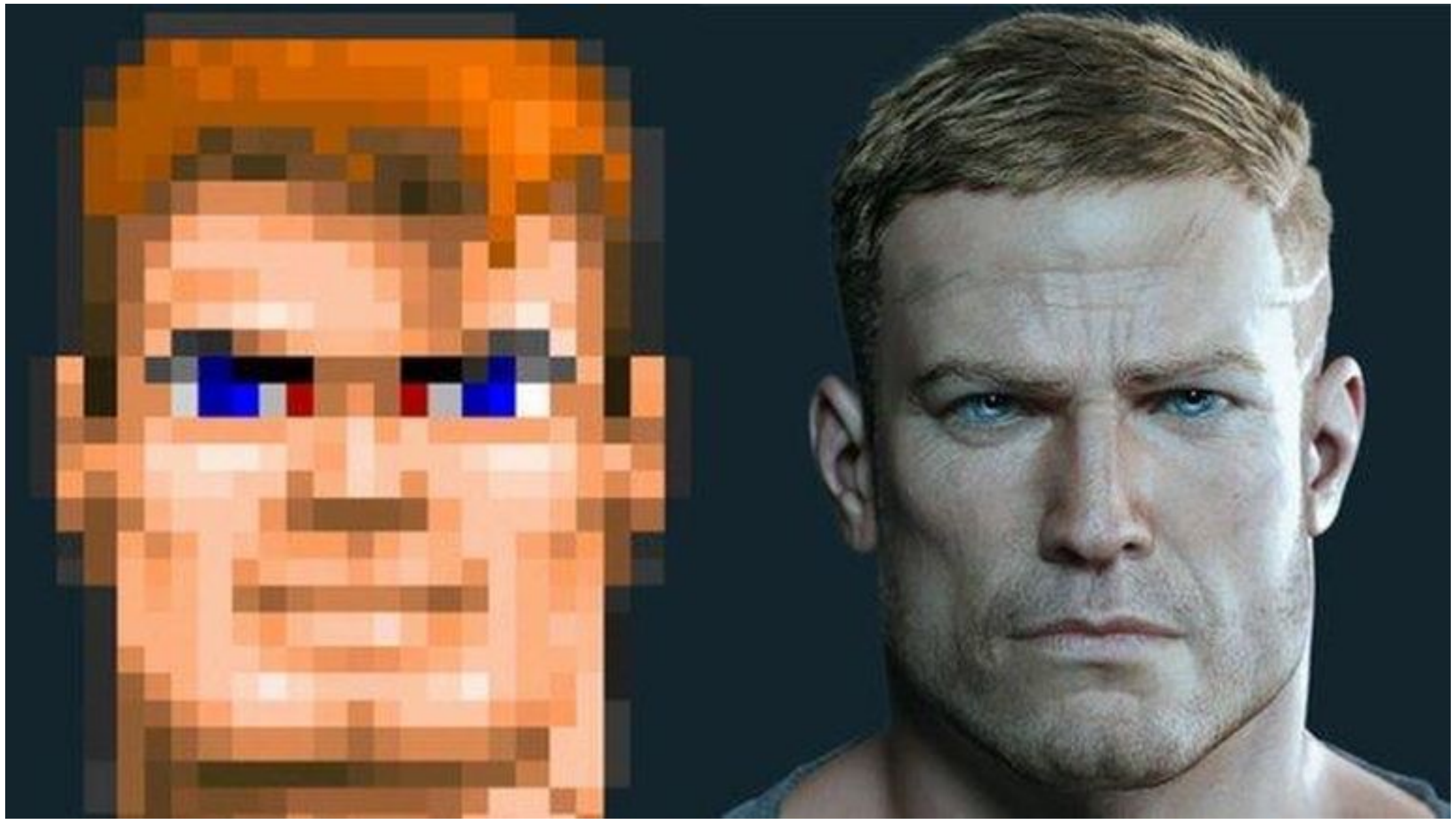
Introduction to GPU computing

1980



Introduction to GPU computing

2020



Introduction to GPU computing

2020

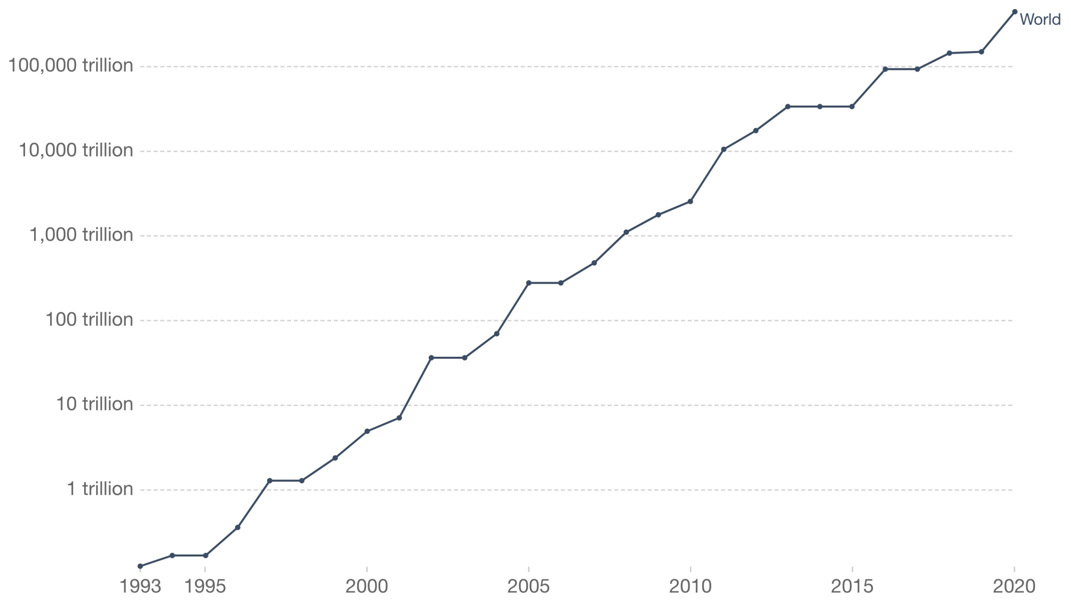
Introduction to GPU computing

Hardware Performance metrics

- Clock frequency [GHz] = absolute hardware speed
    - Memories,CPUs,interconnects
- Operational speed [GFLOPs]
    - Operations per second
      single AND double precision
- Memory bandwidth [GB/s]
    - Memory operations per second
      - Can differ for read and write operations!
    - Differs a lot between different memories on chip
- Power [Watt]

Supercomputer Power (FLOPS), 1993 to 2020

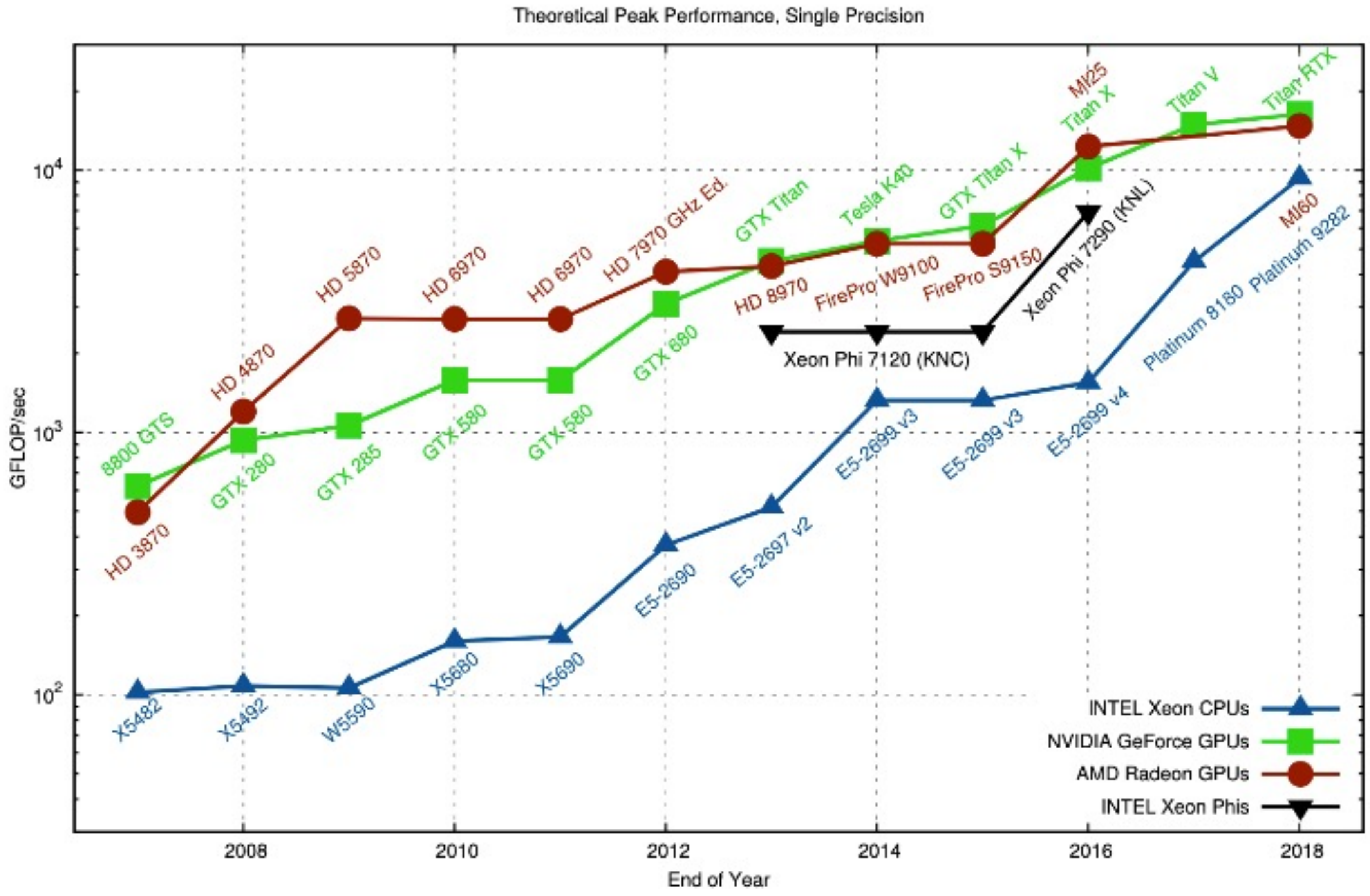
The growth of supercomputer power, measured as the number of floating-point operations carried out per second (FLOPS) by the largest supercomputer in any given year. FLOPS are a measure of calculations per second for floating-point operations. Floating-point operations are needed for very large or very small real numbers, or computations that require a large dynamic range. It is therefore a more accurate measured than simply instructions per second.



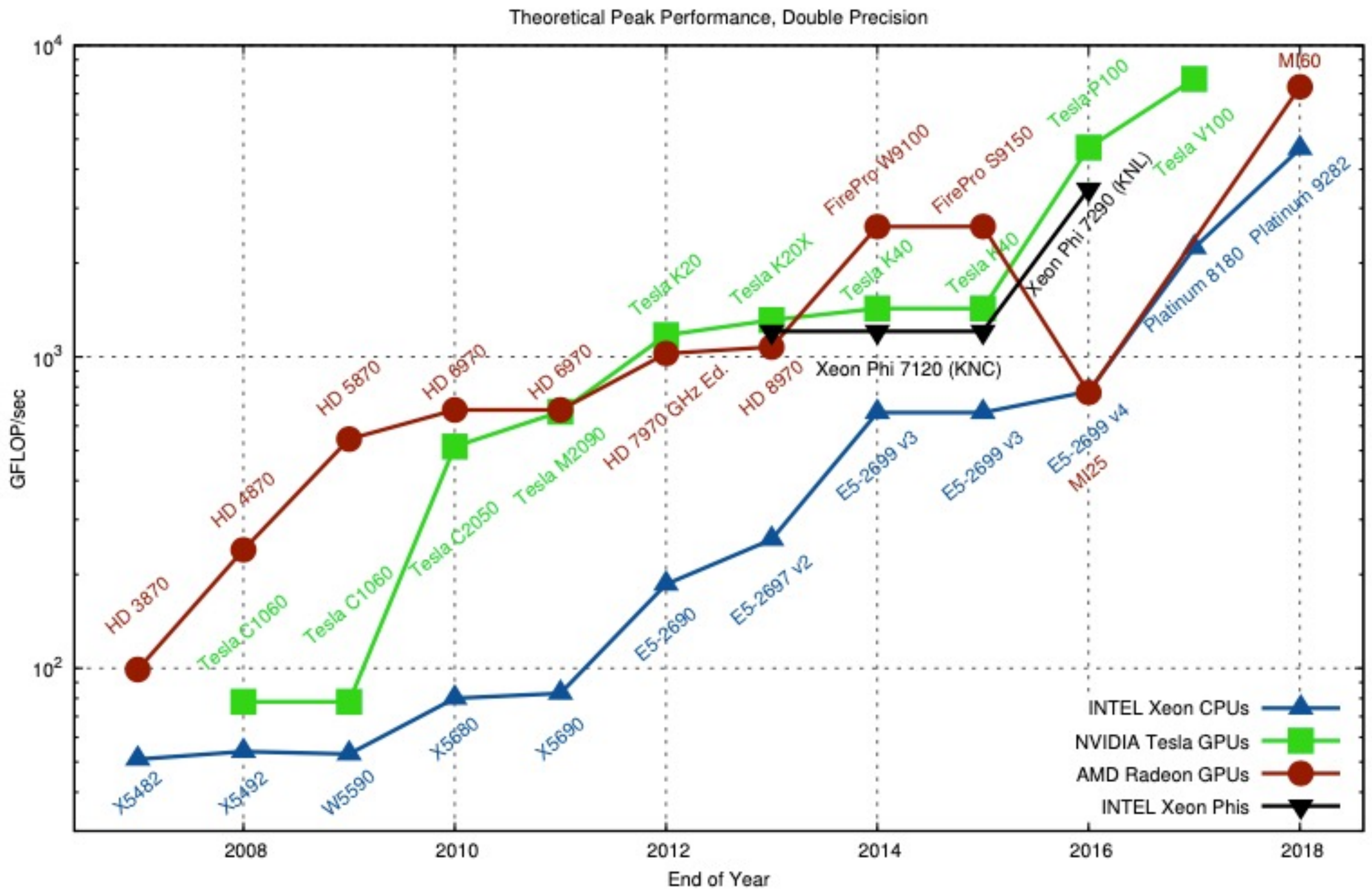
Source: TOP500 Supercomputer Database

CC BY

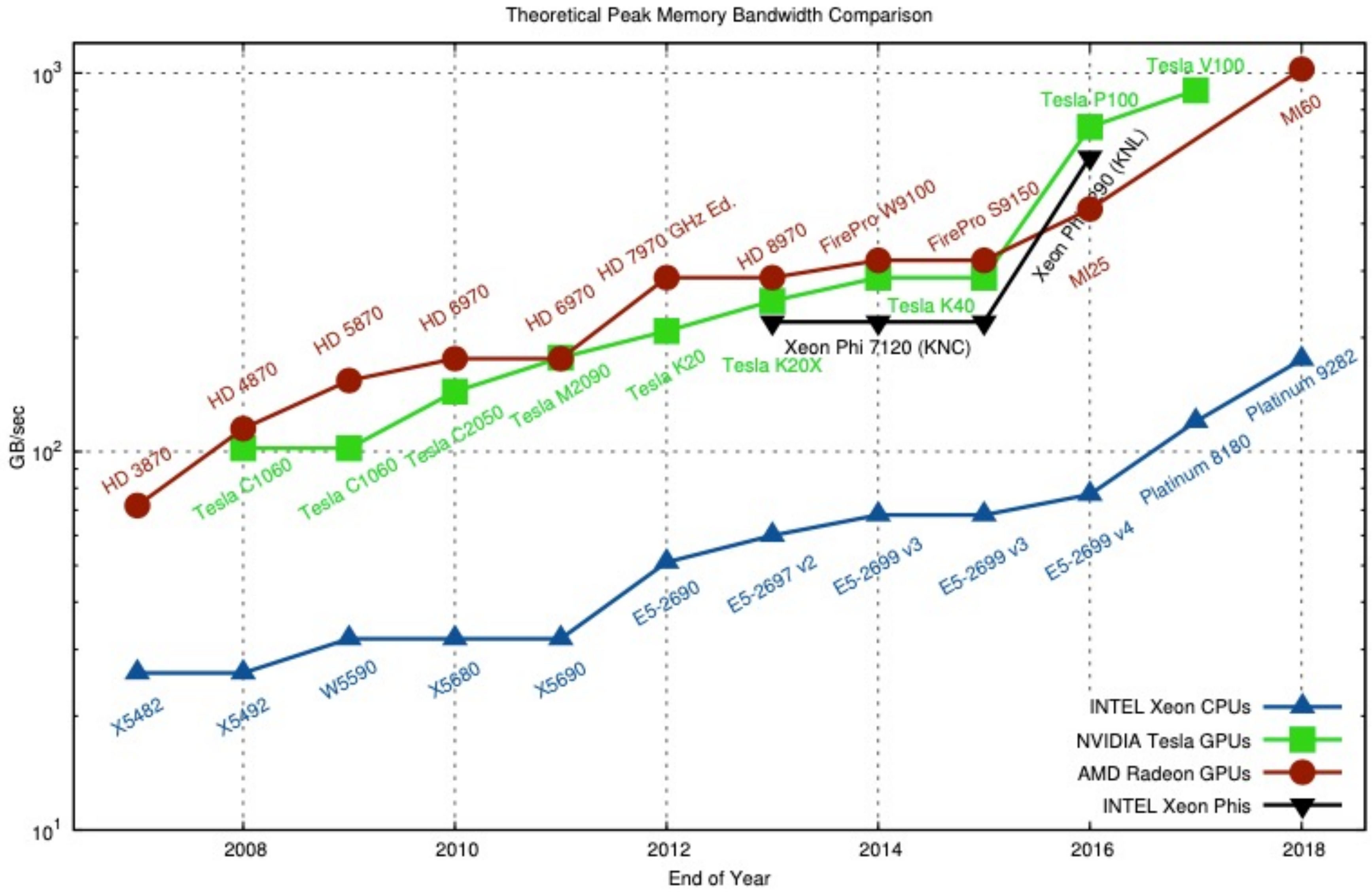
Introduction to GPU computing



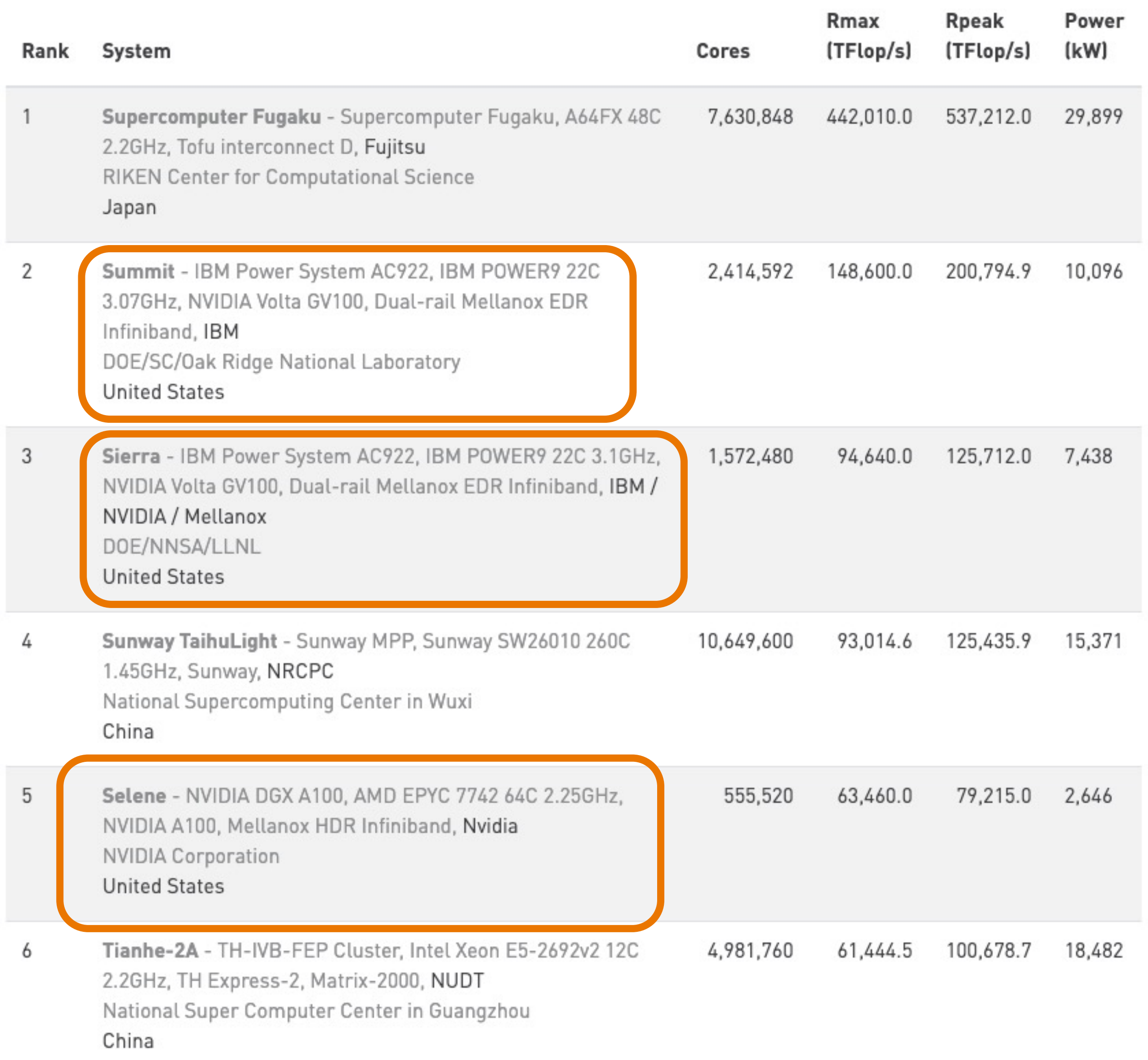
Introduction to GPU computing



Introduction to GPU computing



Introduction to GPU computing



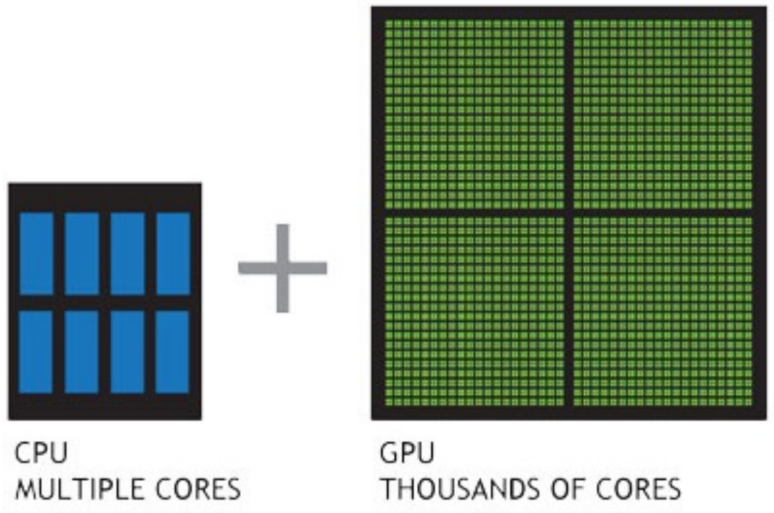
Rank	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442,010.0	537,212.0	29,899
2	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM DOE/SC/Oak Ridge National Laboratory United States	2,414,592	148,600.0	200,794.9	10,096
3	**Sierra** - IBM Power System AC922, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband, IBM / NVIDIA / Mellanox DOE/NNSA/LLNL United States	1,572,480	94,640.0	125,712.0	7,438
4	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway, NRCPC National Supercomputing Center in Wuxi China	10,649,600	93,014.6	125,435.9	15,371
5	**Selene** - NVIDIA DGX A100, AMD EPYC 7742 64C 2.25GHz, NVIDIA A100, Mellanox HDR Infiniband, Nvidia NVIDIA Corporation United States	555,520	63,460.0	79,215.0	2,646
6	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000, NUDT National Super Computer Center in Guangzhou China	4,981,760	61,444.5	100,678.7	18,482

www.top500.org

Introduction to GPU computing

Benefits of using GPUs

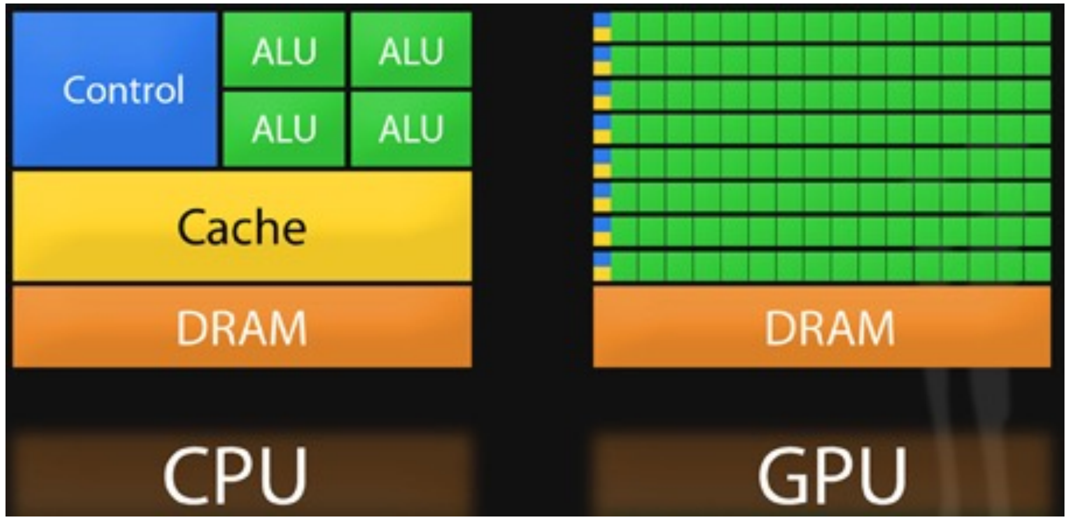
- Graphics processing units (GPUs) were originally designed for running games and graphics workloads that were highly parallel in nature.
- High demand in FLOPS for data parallel throughput workloads
- GPGPU: general purpose computing on GPUs, highly parallel, multithreaded, manycore processor with high computational power and high memory bandwidth.



Introduction to GPU computing

GPU vs CPU

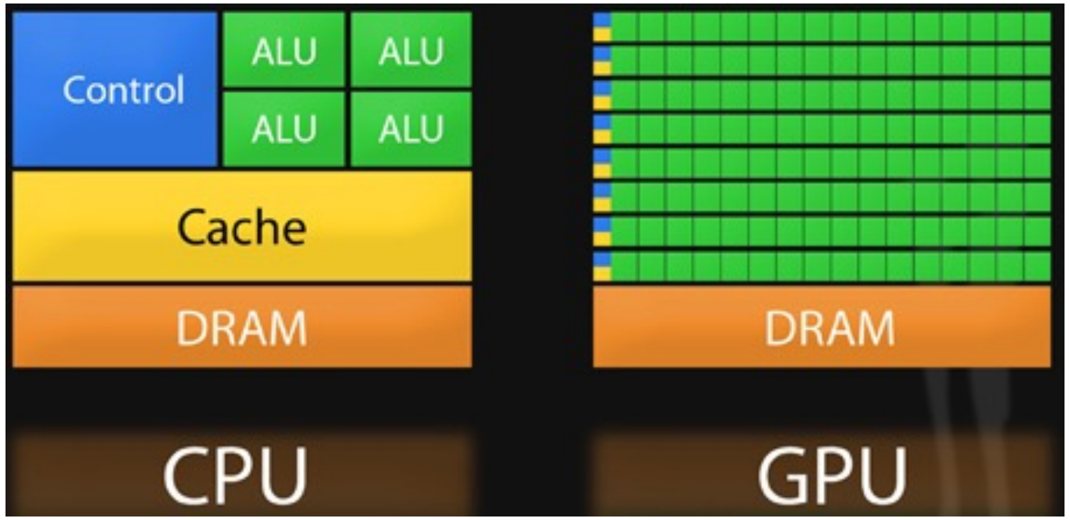
- CPU for ”heavy” tasks with complex control flows and branching (VM, branching, security, etc.)
- GPU do one thing well: handling billions of repetitive ”light” tasks
- GPUs have 1000s of ALU compared to traditional CPUs that are commonly build with 4-8
- GPU is specialized for compute-intensive highly parallel computations with more transistors dedicated to data processing rather than flow control and caching



Introduction to GPU computing

GPU vs CPU

- Different design for different goals
  - GPU workload is highly parallel
  - CPU general purpose
- CPU: minimize latency from one thread
  - Big on-chip chaces
  - control logic
- GPU: maximize throughput
  - Multithreading can hide latency
  - Simple/shared control login



Introduction to GPU computing

GPU vs CPU

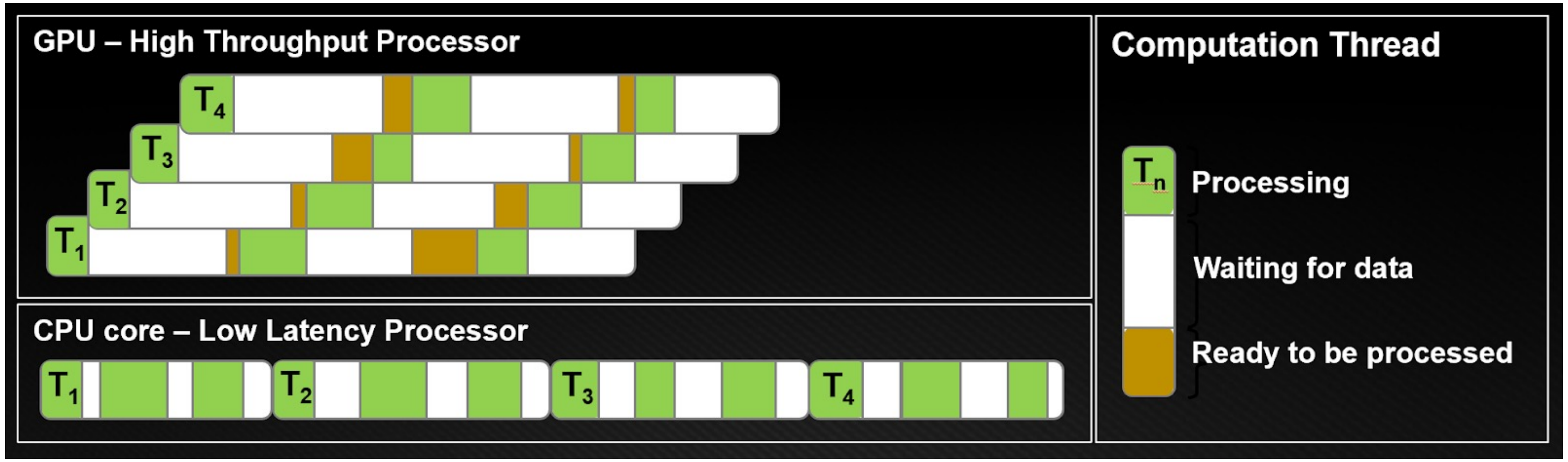


Image source: developer.nvidia.com

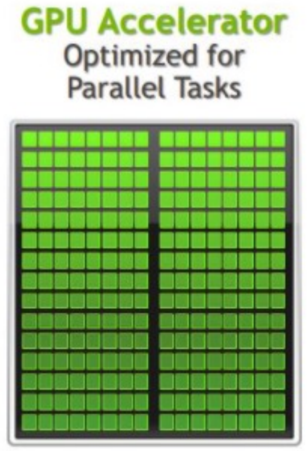
- CPU architectures must minimize latency within each thread
- GPUs, threads are lightweight, so a GPU can switch from stalled threads to other threads at no cost, minimizing latency with computation

Introduction to GPU computing

GPU vs CPU

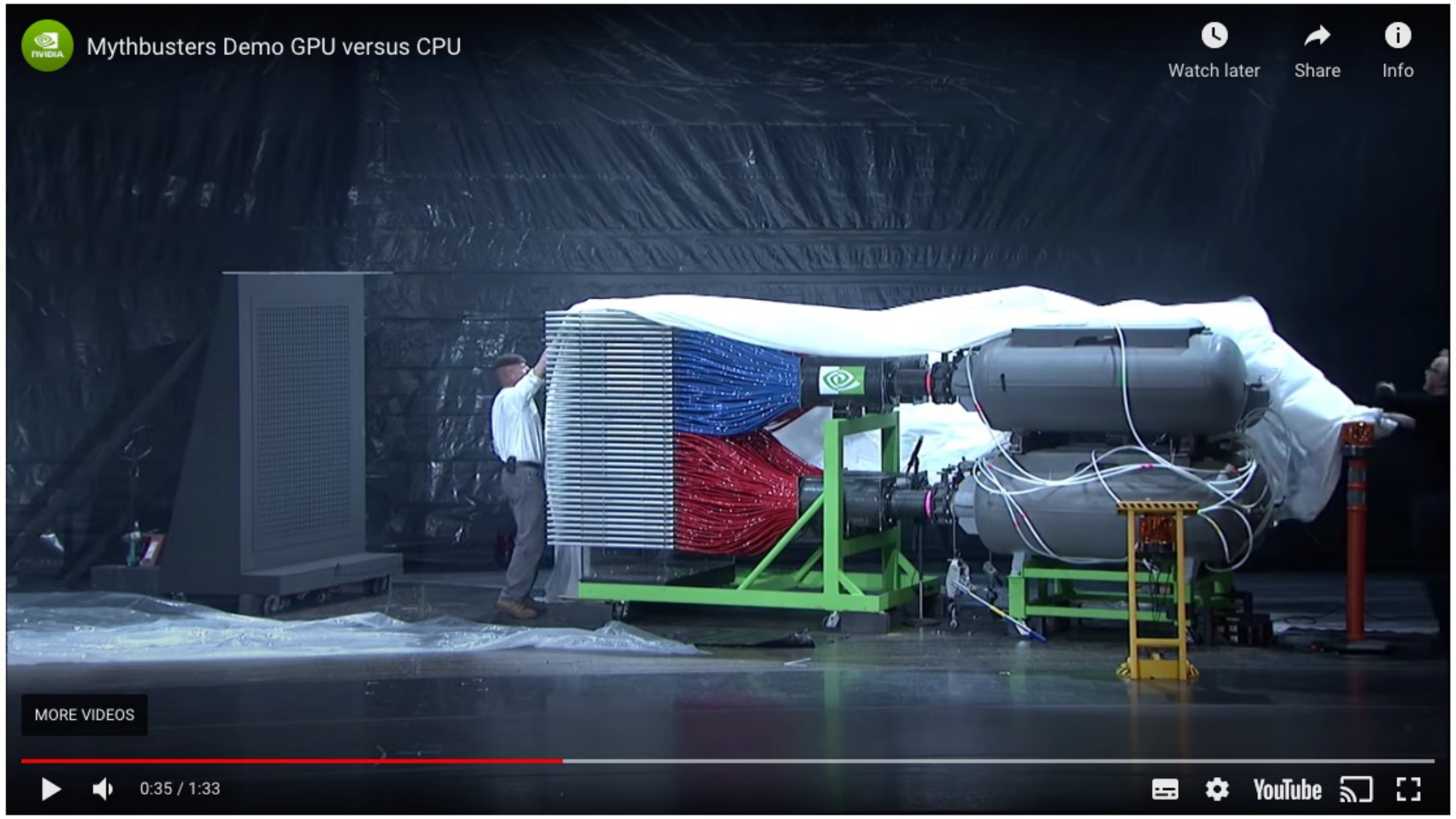
CPU strengths	GPU strengths
Very large main memory	High bandwidth main memory
Very fast clock speeds	Latency tolerant via parallelism
Latency optimized via large caches	Significantly more compute resources
Small number of threads can run very quickly	High throughput
	High performance/watt

CPU weaknesses	GPU weaknesses
Relatively low memory bandwidth	Relatively low memory capacity
Low performance/watt	Low per-thread performance



Introduction to GPU computing

GPU vs CPU

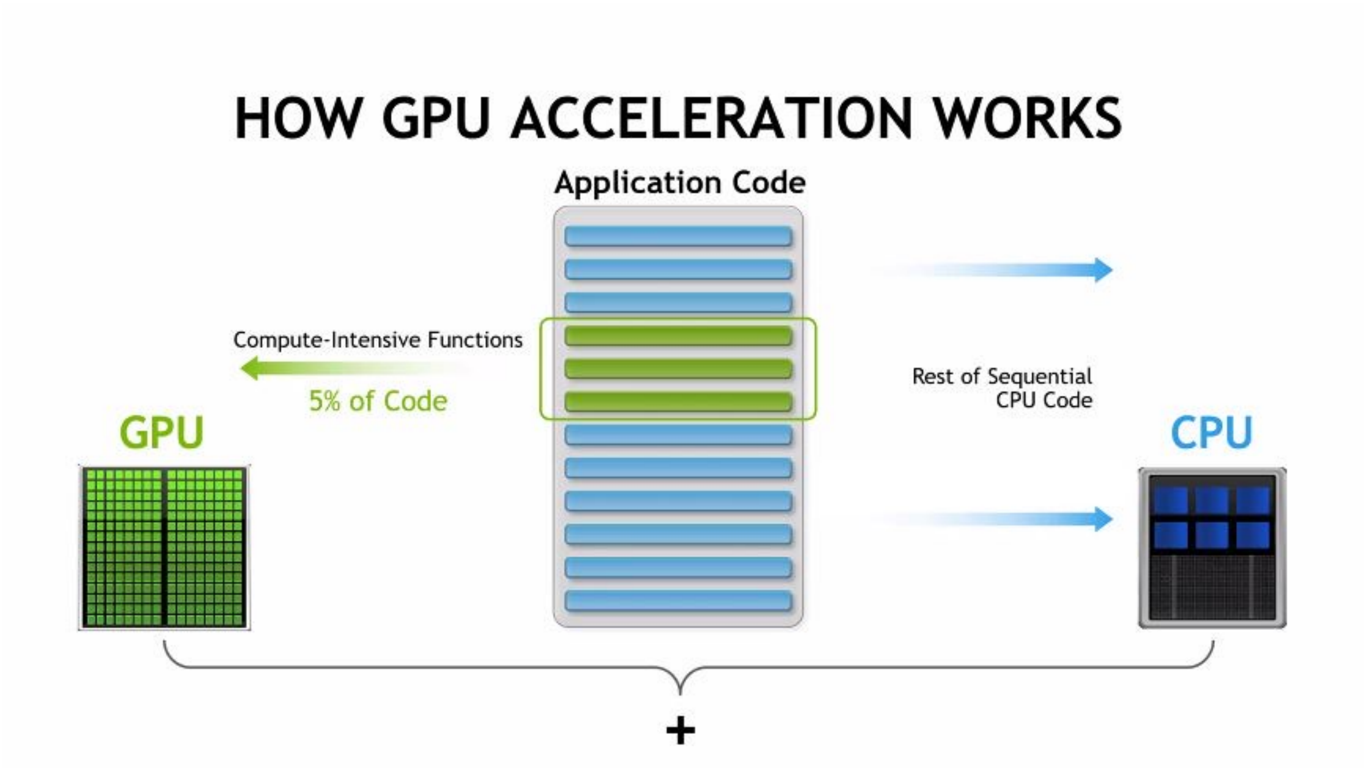


https://youtu.be/-P28LKWTzrI

Introduction to GPU computing

GPU vs CPU

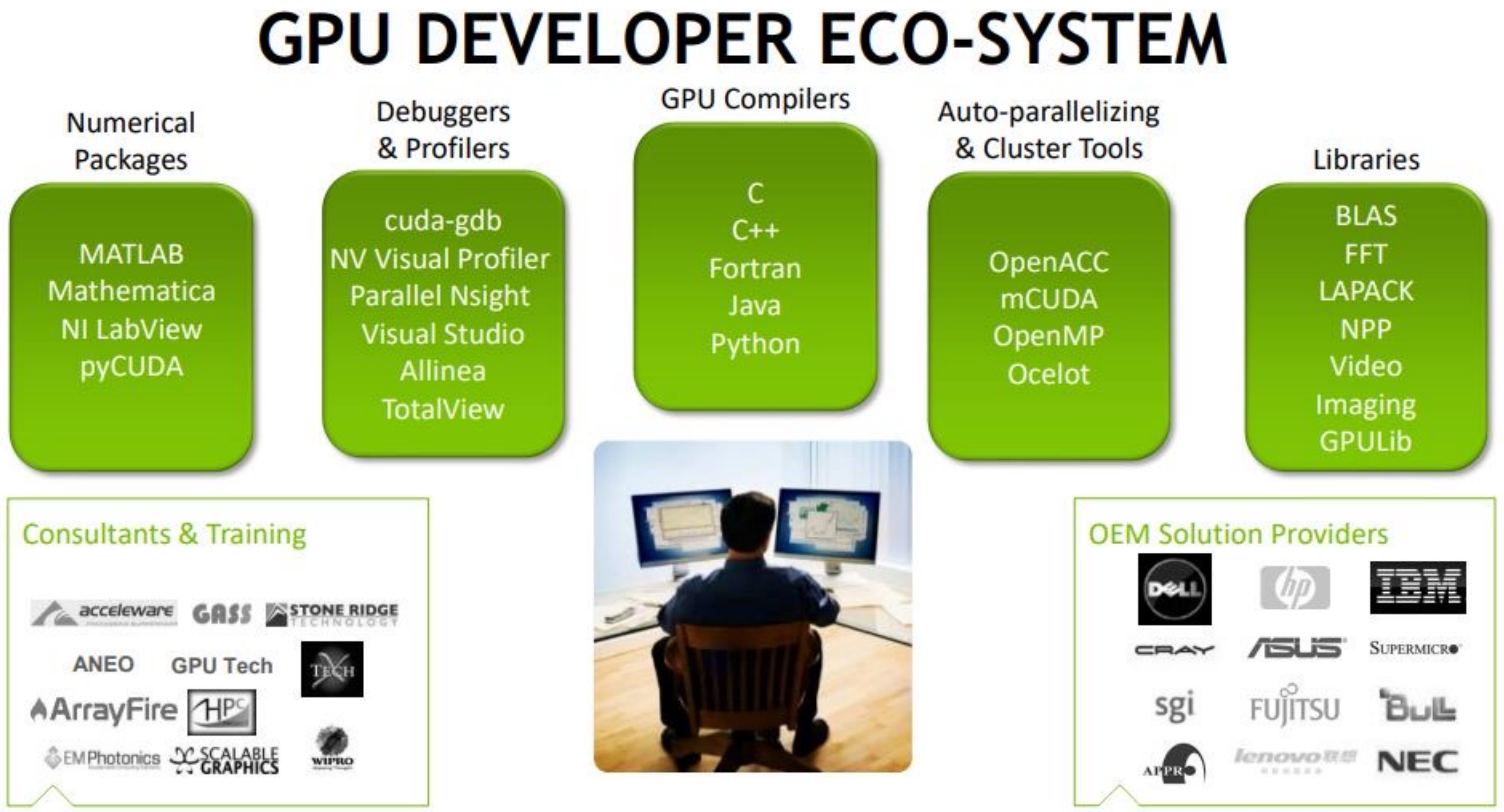
DEVICE


HOW GPU ACCELERATION WORKS
Application Code
Compute-Intensive Functions
5% of Code
GPU
Rest of Sequential CPU Code
CPU
+

HOST

Introduction to GPU computing

GPU ecosystem



GPU DEVELOPER ECO-SYSTEM

Numerical Packages
- MATLAB
- Mathematica
- NI LabView
- pyCUDA

Debuggers & Profilers
- cuda-gdb
- NV Visual Profiler
- Parallel Nsight
- Visual Studio
- Allinea
- TotalView

GPU Compilers
- C
- C++
- Fortran
- Java
- Python

Auto-parallelizing & Cluster Tools
- OpenACC
- mCUDA
- OpenMP
- Ocelot

Libraries
- BLAS
- FFT
- LAPACK
- NPP
- Video
- Imaging
- GPULib

Consultants & Training

acceleware, GASS, STONE RIDGE TECHNOLOGY, ANEO, GPU Tech, TECH-X, ArrayFire, HPC, EM Photonics, SCALABLE GRAPHICS, WIPRO

OEM Solution Providers

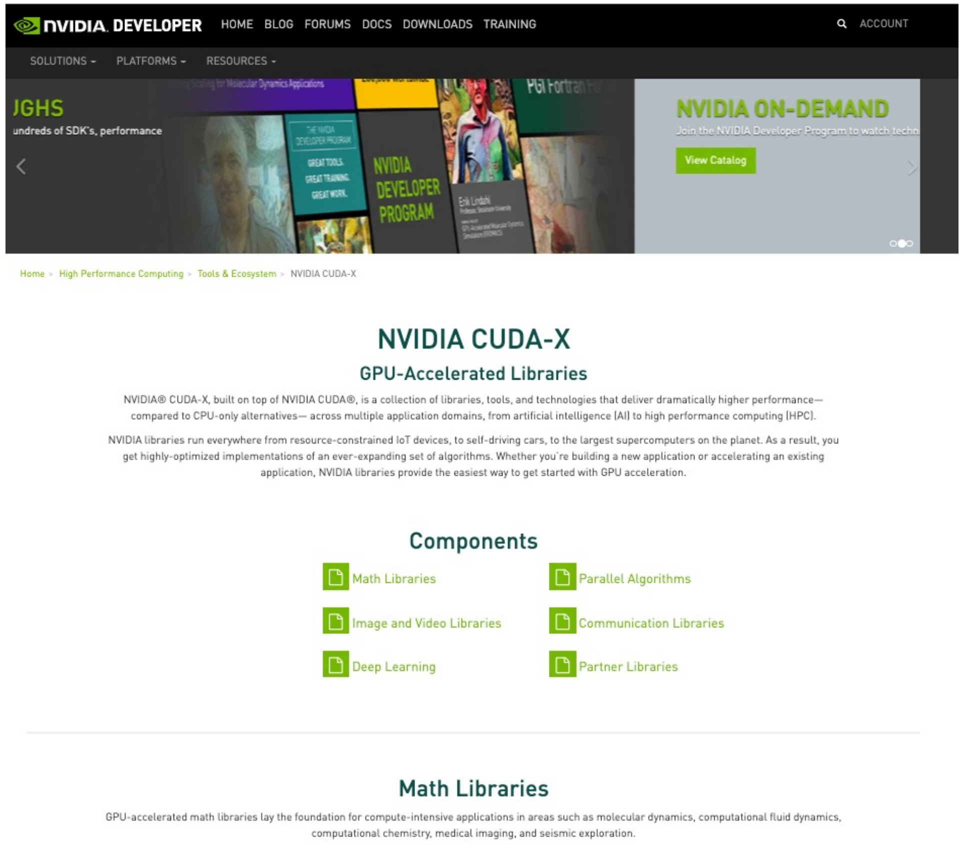
DELL, hp, IBM, CRAY, ASUS, SUPERMICRO, sgi, FUJITSU, BULL, APPRO, lenovo, NEC

Introduction to GPU computing

GPU ecosystem

CUDA Tools and Ecosystem described in details on the NVIDIA Developer Zone
https://developer.nvidia.com/

CUDA Tools and Ecosystem described in details on the NVIDIA Developer Zone
https://developer.nvidia.com/gpu-accelerated-libraries


NVIDIA DEVELOPER
HOME BLOG FORUMS DOCS DOWNLOADS TRAINING
ACCOUNT
SOLUTIONS PLATFORMS RESOURCES
JGHS
undreds of SDK's, performance
NVIDIA DEVELOPER PROGRAM
NVIDIA ON-DEMAND
Join the NVIDIA Developer Program to watch techn
View Catalog
Home > High Performance Computing > Tools & Ecosystem > NVIDIA CUDA-X

NVIDIA CUDA-X

GPU-Accelerated Libraries

NVIDIA® CUDA-X, built on top of NVIDIA CUDA®, is a collection of libraries, tools, and technologies that deliver dramatically higher performance—compared to CPU-only alternatives— across multiple application domains, from artificial intelligence (AI) to high performance computing (HPC).

NVIDIA libraries run everywhere from resource-constrained IoT devices, to self-driving cars, to the largest supercomputers on the planet. As a result, you get highly-optimized implementations of an ever-expanding set of algorithms. Whether you're building a new application or accelerating an existing application, NVIDIA libraries provide the easiest way to get started with GPU acceleration.

Components

- Math Libraries
- Parallel Algorithms
- Image and Video Libraries
- Communication Libraries
- Deep Learning
- Partner Libraries

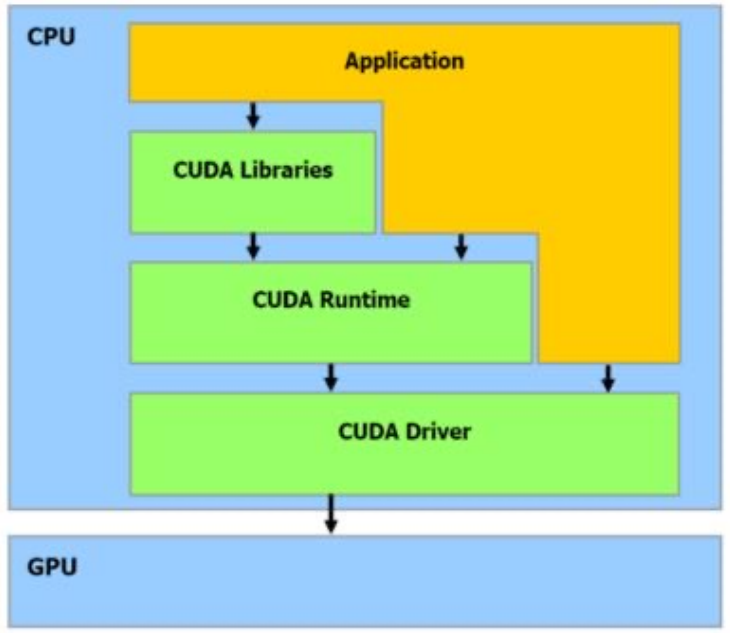
Math Libraries

GPU-accelerated math libraries lay the foundation for compute-intensive applications in areas such as molecular dynamics, computational fluid dynamics, computational chemistry, medical imaging, and seismic exploration.

Introduction to GPU computing

Introduction to CUDA programming model

- Introduced by NVIDIA in 2006, Compute Unified Device Architecture
- General purpose programming model that leverages the parallel compute engine in NVIDIA GPUs
- Targeted software stack
- Driver for loading computation programs into GPU:
    - standalone driver optimized for computation
    - explicit GPU memory managemnt



Introduction to GPU computing

Introduction to CUDA programming model

The GPU is viewed as a compute device that:

- Is a coprocessor to the CPU (or host)
- Has its own DRAM (device memory)
- Runs many threads in parallel
  - Hardware switching between threads fast
  - Overprovision (1000s of threads) hide latencies
- Data-parallel portions of an application are executed on the device as **kernels** which run in parallel on many threads
- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few

Introduction to GPU computing

Introduction to CUDA programming model

To execute any CUDA program, there are three main steps:

1. Copy the input data from host memory to device memory, also known as host-to-device transfer.

2. Load the GPU program and execute, caching data on-chip for performance.

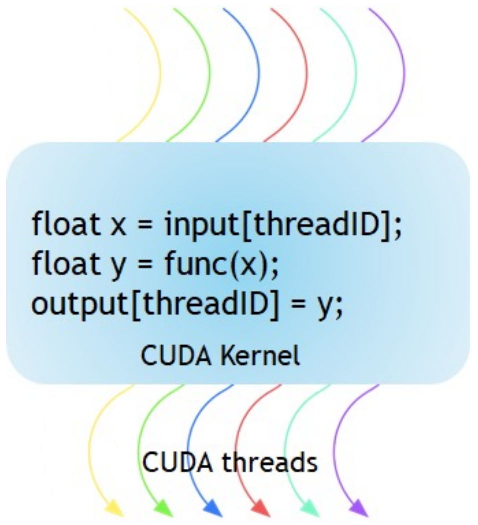
3. Copy the results from device memory to host memory, also called device-to-host transfer.

Introduction to GPU computing

Introduction to CUDA programming model

CUDA kernel and thread hierarchy

- The CUDA kernel is the portion of code executed on the GPU
- This is executed *n* times in parallel by *n* different CUDA threads
- A group of threads is called “threads block” and they are grouped into “Grids”
- A kernel is executed as a grid of blocks of threads


```
float x = input[threadID];
float y = func(x);
output[threadID] = y;
```

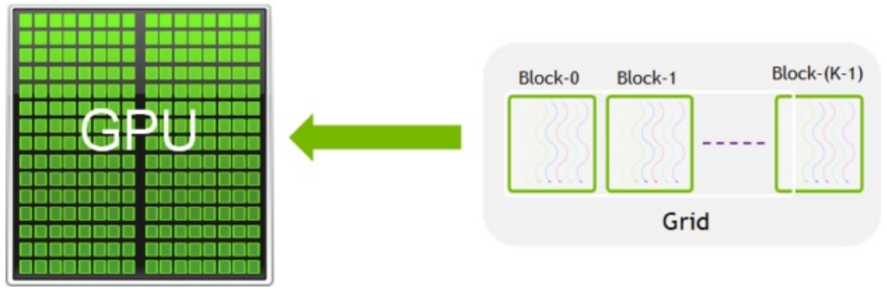


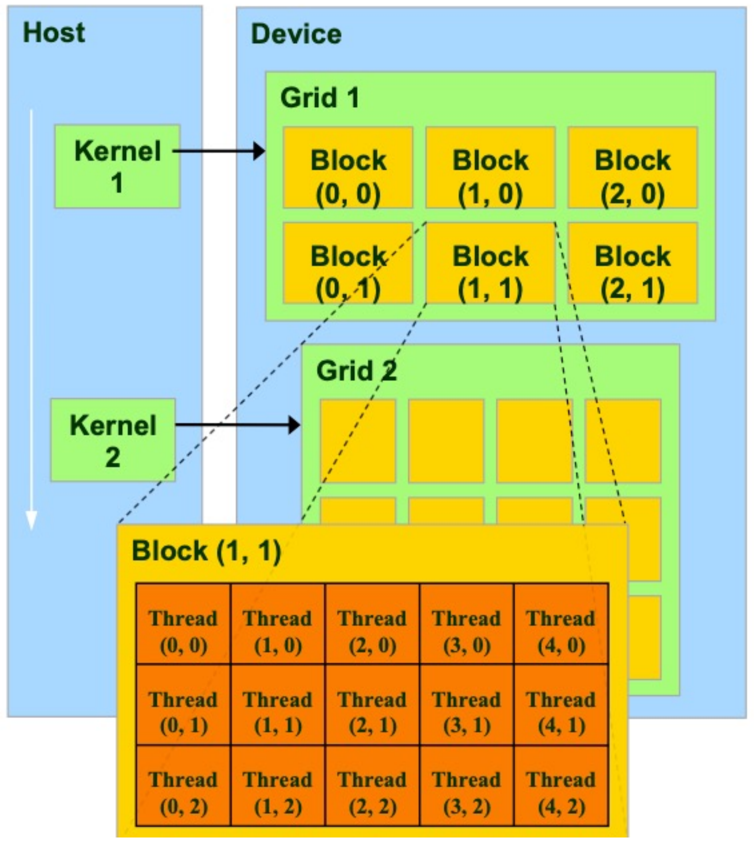
Image source: developer.nvidia.com

Introduction to GPU computing

Introduction to CUDA programming model

CUDA kernel and thread hierarchy

- CUDA defines built-in 3D variables for threads and blocks (threadIdx, blockIdx, blockDim)
- Threads in the same block can communicate with each other via shared memory, barrier synchronization or other synchronization primitives such as atomic operations.
- The number of threads per block and the number of blocks per grid specified at call time of the kernel by the programmer.
- CUDA architecture limits the numbers of threads per block (1024 threads per block limit).



Introduction to GPU computing

Introduction to CUDA programming model

```
// Kernel – Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float MatB[N][N], float MatC[N][N])
{
            int i = blockIdx.x * blockDim.x + threadIdx.x;
            int j = blockIdx.y * blockDim.y + threadIdx.y;
            if (i < N && j < N)
            MatC[i][j] = MatA[i][j] + MatB[i][j];
}

int main()
{
            ...
            // Matrix addition kernel launch from host code
            dim3 threadsPerBlock(16, 16);
            dim3 numBlocks((N + threadsPerBlock.x -1) / threadsPerBlock.x, (N+threadsPerBlock.y -1)/ threadsPerBlock.y);
            MatAdd<<<numBlocks, threadsPerBlock>>>(MatA, MatB, MatC);
            ...
}
```

Introduction to GPU computing

Introduction to CUDA programming model

```
// Kernel – Adding two matrices MatA and MatB
__global__ void MatAdd(float MatA[N][N], float MatB[N][N], float MatC[N][N])
{
            int i = blockIdx.x * blockDim.x + threadIdx.x;
            int j = blockIdx.y * blockDim.y + threadIdx.y;
            if (i < N && j < N)
            MatC[i][j] = MatA[i][j] + MatB[i][j];
}

int main()
{
            ...
            // Matrix addition kernel launch from host code
            dim3 threadsPerBlock(16, 16);
            dim3 numBlocks((N + threadsPerBlock.x -1) / threadsPerBlock.x, (N+threadsPerBlock.y -1)/ threadsPerBlock.y);
            MatAdd<<<numBlocks, threadsPerBlock>>>(MatA, MatB, MatC);
            ...
}
```

Introduction to GPU computing

Introduction to CUDA programming model

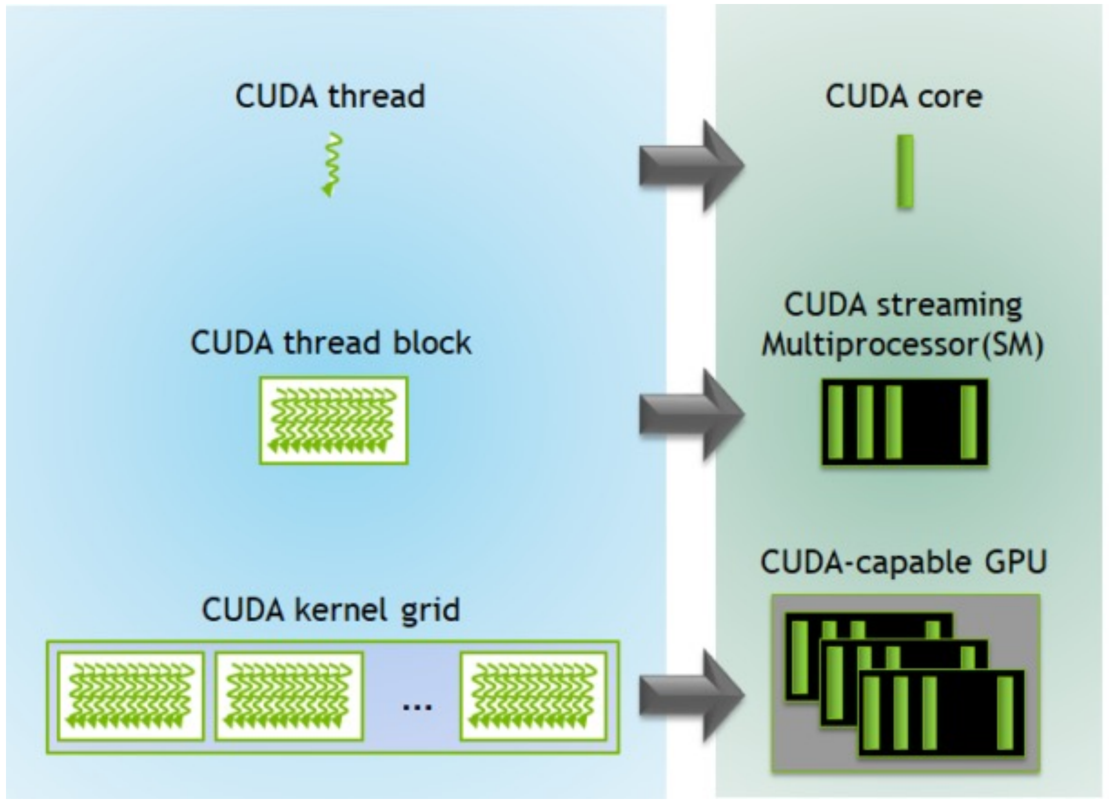
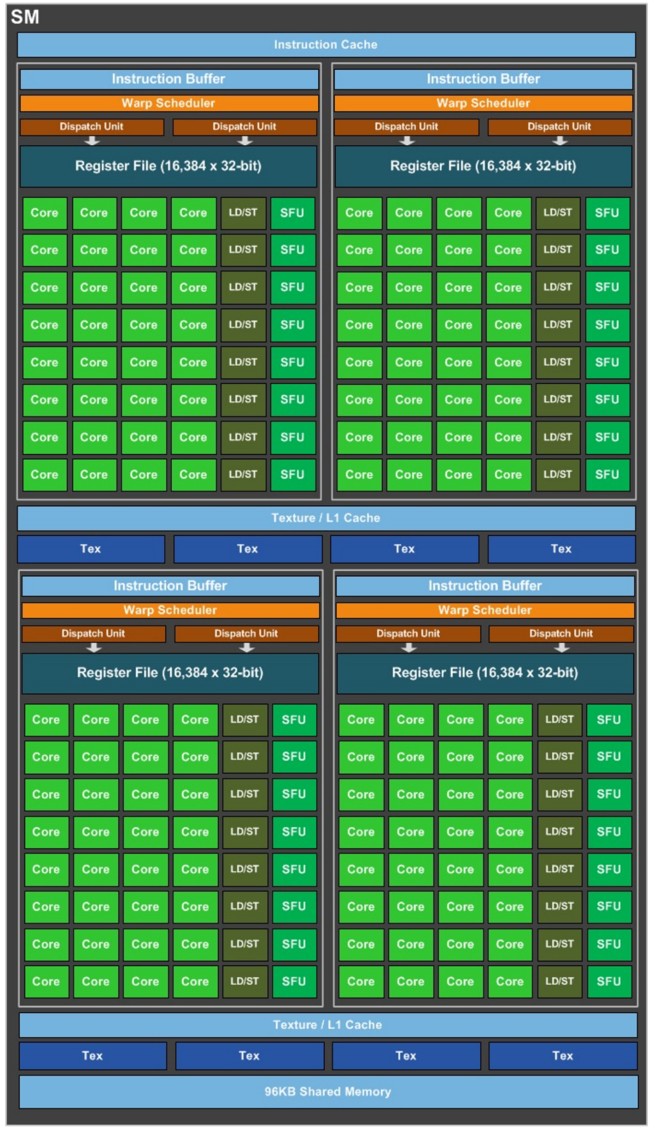


Image source: developer.nvidia.com

Introduction to GPU computing

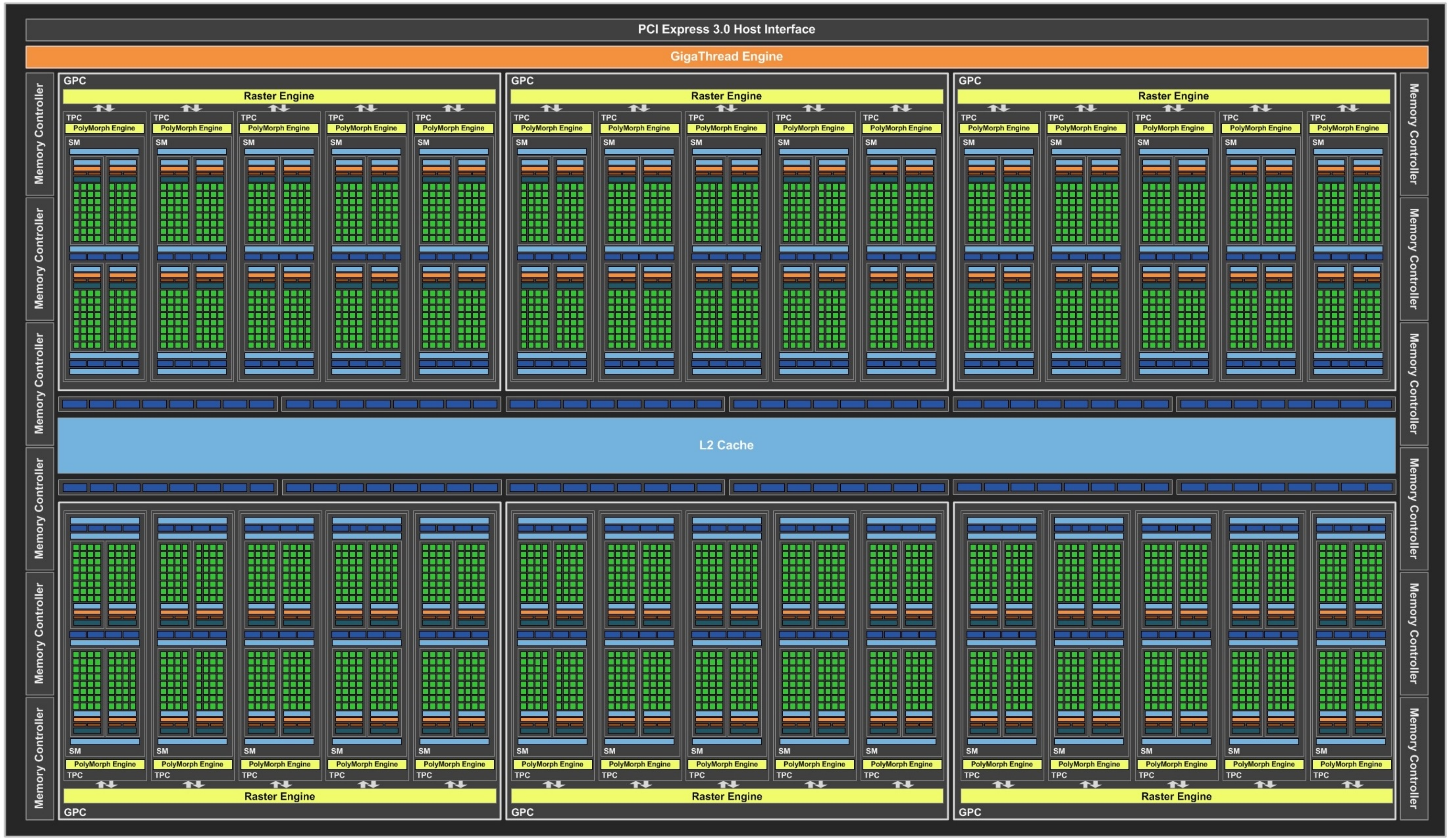
GPU architecture

- The hardware groups threads that execute the same instruction into warps.
- Several warps constitute a thread block.
- Several thread blocks are assigned to a Streaming Multiprocessor (SM).
- Several SM constitute the whole GPU unit (which executes the whole Kernel Grid).



Introduction to GPU computing

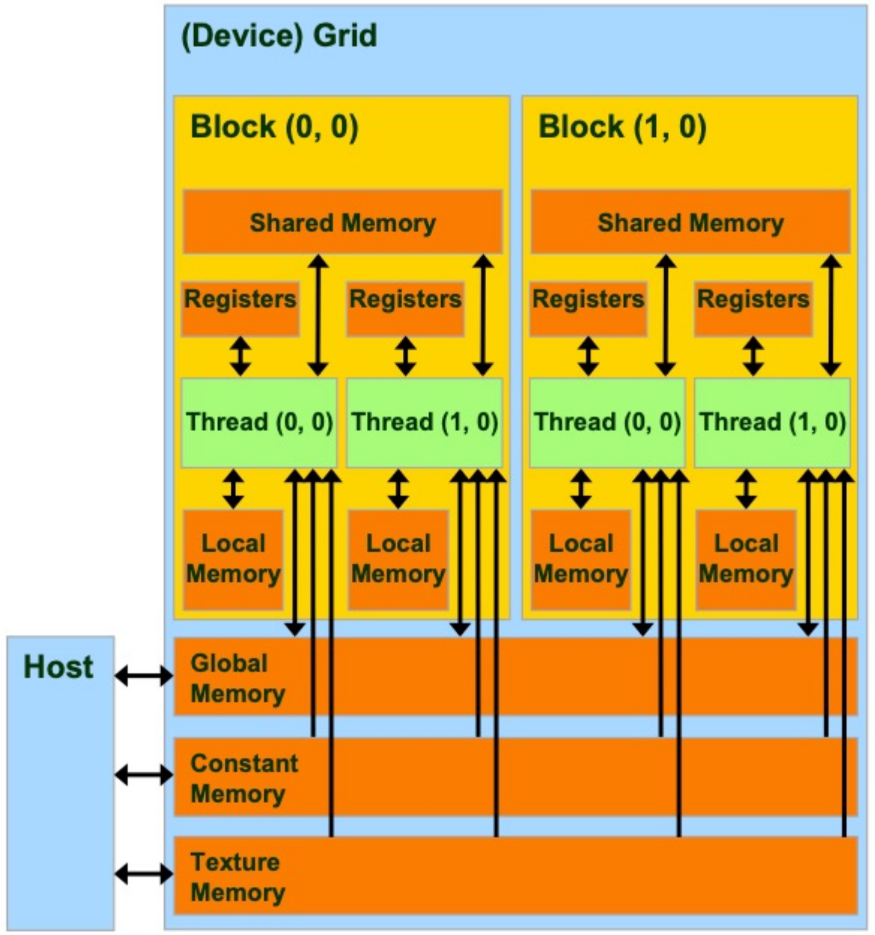
GPU architecture



Introduction to GPU computing

GPU architecture

- **Global memory** This memory is accessible to all threads as well as the host (CPU) and is allocated and deallocated by the host. Used to initialize the data that the GPU will work on.
- **Shared memory** Each *thread block* has its own shared memory which is accessible only by threads within the block. Much faster than global memory. Requires special handling to get maximum performance and only exists for the lifetime of the block.
- **Local memory** Each *thread* has its own private local memory. Only exists for the lifetime of the thread and is generally handled automatically by the compiler.
- **Constant and texture memory** are read-only memory spaces accessible by all threads. Constant memory is used to cache values that are shared by all functional units Texture memory is optimized for texturing operations provided by the hardware



Introduction to GPU computing

Introduction to CUDA programming model

There are different ways to optimize CUDA codes:

- Number of threads per block
- Workload per thread
- Total work per thread block
- Correct memory access and data locality
- …

Introduction to GPU computing

PyCUDA

PyCUDA gives you easy, Pythonic access
to Nvidia’s CUDA parallel computation API.

https://documen.tician.de/pycuda/

Numba

Numba is an open source JIT compiler that translates a
subset of Python and NumPy code into fast machine code.

http://numba.pydata.org/